# Parser Bottom-Up

(o Shift-Reduce)

- shift: un simbolo terminale viene spostato dall'input sulla pila

- reduce: una serie di simboli (terminali e non terminali) sulla cima della pila corrisponde al "reverse" di una parte destra di una produzione

  $A \to \alpha \in R$ — $\alpha^R$ sulla pila

  La stringa $\alpha^R$ viene rimossa dalla pila e sostituita con $A$

  ("$\alpha$ viene ridotta ad $A$")

Riprendiamo la presentazione di un parser shift-reduce nondeterministico, che è essenzialmente un PDA con un solo stato che riconosce per pila vuota il linguaggio $L \cdot \#$

Parser LR: - L (leggo da sx a dx)

- R (derivazione rightmost)

# Parser shift-reduce Nondeterministico

**Input**: - una grammatica libera $G$ con simbolo iniziale $S$
- una stringa $w \in T^*$

**Output**: se $w \in L(G)$, allora restituisce la sua derivazione rightmost a rovescio

- Inizializziamo la pila a \$;
- Inizializziamo l'input con $w$\$;
- Usiamo il PDA seguente per trovare la derivazione per $w$\$

$$M = (T, \{q\}, \underbrace{T \cup N \cup \{\$\}}_{\Gamma}, \delta, \$, \emptyset)$$

dove

1. $(q, aX) \in \delta(q, a, X)$ $\forall a \in T$ $\forall X \in \Gamma$ (SHIFT)
2. $(q, A) \in \delta(q, \varepsilon, \alpha^R)$ se $A \to \alpha \in R$ (REDUCE)
3. $(q, \varepsilon) \in \delta(q, \$, S\$)$ (Accept)

- ogni volta che facciamo "Reduce", forniamo in output la produzione usata
- alla fine, $S$\$ sulla pila, con \$ in input.
  $\Rightarrow$ ok, accettiamo

Oss: generalizzazione della def. di PDA dove non si consuma solo il top della pila, ma una serie di caratteri contigui a cominciare dal top

$E \to T \mid T+E \mid T-E$

$T \to A \mid A*T$

$A \to a \mid b \mid (E)$

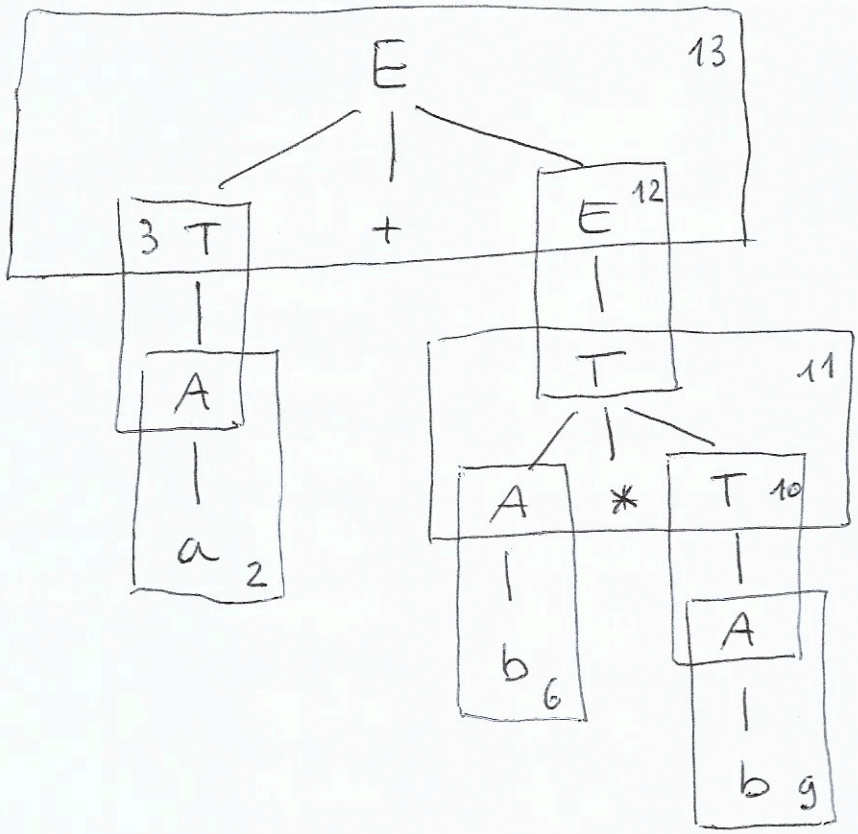
| | Pila → | Input | Azione | Output |
|---|---|---|---|---|
| 1. | $ | a+b*b$ | shift | |
| 2. | $a | +b*b$ | reduce | $A \to a$ |
| 3. | $A | +b*b$ | reduce | $T \to A$ |
| 4. | $T | +b*b$ | shift | |
| 5. | $T+ | b*b$ | shift | |
| 6. | $T+b | *b$ | reduce | $A \to b$ |
| 7. | $T+A | *b$ | shift | |
| 8. | $T+A* | b$ | shift | |
| 9. | $T+A*b | $ | reduce | $A \to b$ |
| 10. | $T+A*A | $ | reduce | $T \to A$ |
| 11. | $T+A*T | $ | reduce | $T \to A*T$ |
| 12. | $T+T | $ | reduce | $E \to T$ |
| 13. | $T+E | $ | reduce | $E \to T+E$ |
| 14. | $E | $ | accept | |
| 15. | ε | ε | | |

→
↑

facciamo crescere la pila verso destra, così leggo $\alpha$ (da sinistra a destra), invece di $\alpha^R$ (da destra a sinistra) sulla pila per $A \to \alpha$

$E \Rightarrow T+E \Rightarrow T+T \Rightarrow T+A*T \Rightarrow T+A*A$

$\Rightarrow T+A*b \Rightarrow T+b*b \Rightarrow A+b*b \Rightarrow a+b*b$




- costruzione dell'albero di derivazione bottom-up
- derivazione canonica destra a rovescio
- enorme nondeterminismo:
  - conflitti shift-reduce

    2') $ a +b*b $ shift

    3') $ a+ b*b $
  - conflitti reduce-reduce

    11') $ T+A*T $ reduce $E \to T$

    12') $ T+A*E $

# Come risolvere i conflitti?

**Strategia**: bisogna scegliere l'azione giusta in modo che sulla pila ci sia un **prefisso viabile**

**Prefisso viabile**: è una sequenza $\in (T \cup NT)^*$ che può apparire sulla pila di un parser bottom-up per una computazione che accetta un input

In pratica ... bisogna che la parte top della pila sia un prefisso di una parte dx di una produzione

Nei 2 esempi precedenti:

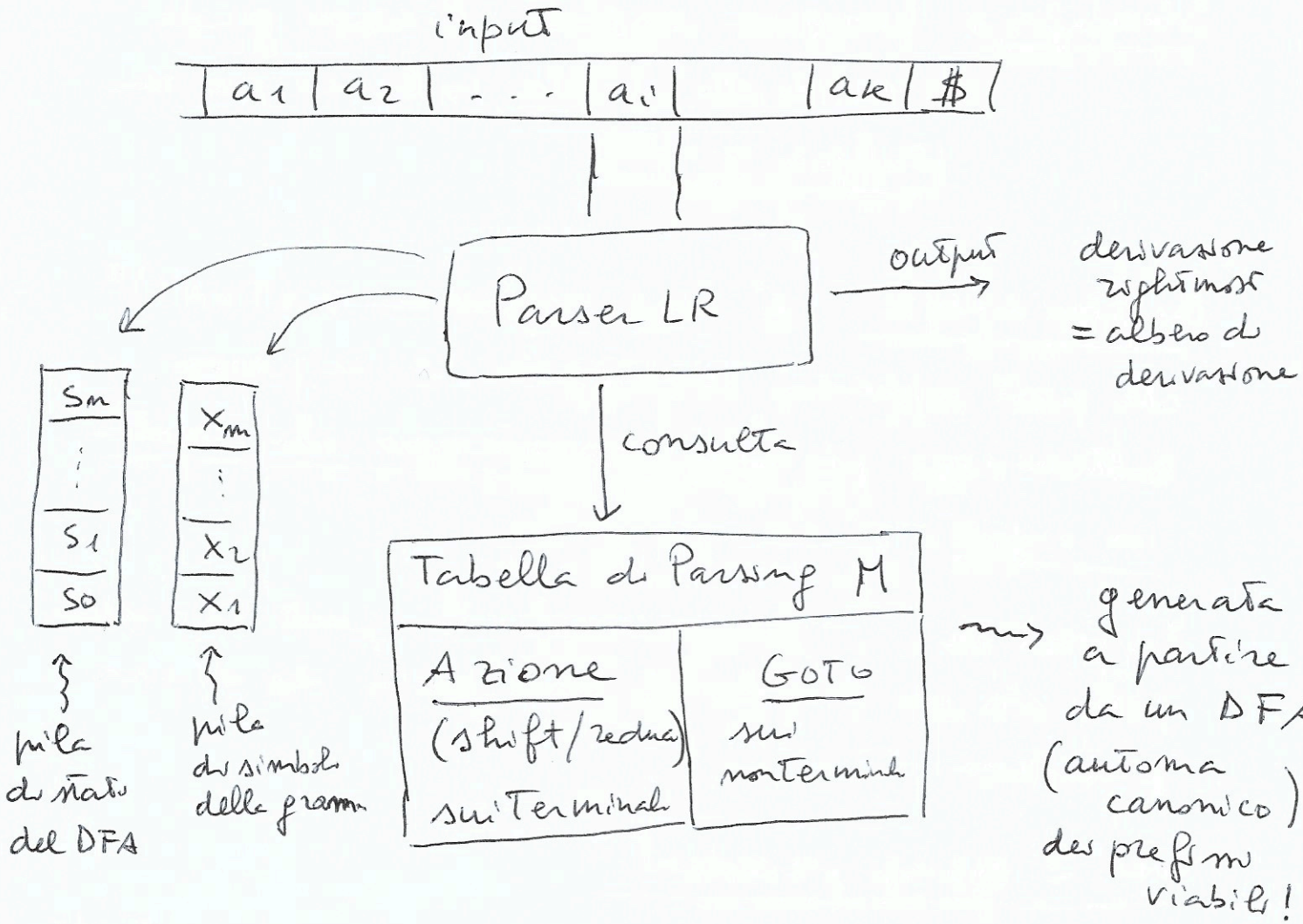
- $ a +  non è un prefisso di parte dx di una produzione, mentre
  $ A  lo è!

- $ T + A * E  non è prefisso di una parte dx di una produzione, mentre
  $ T + T  lo è!

---

Quindi dovremo fornire al PDA, per renderlo deterministico, una struttura di controllo (**tabella di parsing**) che ci aiuti a scegliere l'azione giusta.

# Come è fatto un Parser LR ?




Una <u>configurazione</u> di un parser LR è

$$(\underbrace{s_0 \dots s_m}_{\text{stack degli stati}}, \underbrace{X_1 \dots X_m}_{\text{stack dei simboli}}, \underbrace{a_i \dots a_k \$}_{\text{resto dell'input}})$$

<u>Oss</u>: "$X_1 \dots X_m a_i \dots a_k$" è una stringa intermedia della derivazione canonica destra

# Mosse del Parser LR

(1) Prima legge: [stato top] $s_n$ e [simbolo corrente dell'input] $a_i$

(2) Consulta la tabella di parsing LR $M[s_n, a_i]$

- se $M[s_n, a_i]$ = shift s, allora la nuova configurazione è

$$(s_0 \dots s_n \underline{s}, X_1 \dots X_m \underline{a_i}, a_{i+1} \dots a_k \$)$$

- se $M[s_n, a_i]$ = reduce $A \to \beta$, allora la nuova configurazione è

$$(s_0 \dots s_{n-r} \underline{s}, X_1 \dots X_{m-r} A, a_i \dots a_k \$)$$

dove $r = |\beta|$ e $M[s_{n-r}, A]$ = goto s

cioè fa tre passi:

(1) faccio "pop" di r elementi dai 2 stack

(2) metto A in cima alla pila dei simboli

(3) calcolo il nuovo stato top, guardando $M[s_{n-r}, A]$ = goto s e metto s in cima alla pila degli stati

- se $M[s_n, a_i]$ = accept ⇒ FINE!

- se $M[s_n, a_i]$ = "bianco" ⇒ errore!

# Un esempio

G

| |
|---|
| (1) $S' \to S$ |
| (2) $S \to (S)$ |
| (3) $S \to ()$ |

← grammatica aumentata con un nuovo simbolo iniziale! (assunzione che faremo sempre per parser LR!)

generata a partire dal DFA dei prefissi viabili → **Tabella di Parsing LR(0)** ← non uso look-ahead!

| Stato | AZIONE | | | GOTO |
|---|---|---|---|---|
| | ( | ) | $ | S |
| 0 | s2 | | | g1 |
| 1 | | | Accept | |
| 2 | s2 | s5 | | g3 |
| 3 | | s4 | | |
| 4 | r2 | r2 | r2 | |
| 5 | r3 | r3 | r3 | |

s2 = shift stato 2

r3 = reduce $S \to ()$

g1 = goto 1

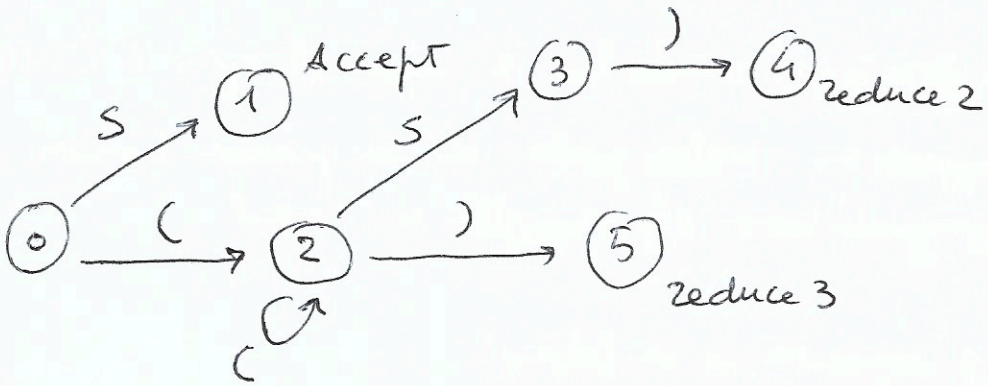
| Stack stati, stack simboli | input | Azione | Output |
|---|---|---|---|
| (0, ε, | (())$ ) | s2 | – |
| (02, (, | ())$ ) | s2 | – |
| (022, ((, | ))$ ) | s5 | – |
| (0225, (((), | )$ ) | r3 | $S \to ()$ |
| (023, (S, | )$ ) | s4 | – |
| (0234, (S), | $ ) | r2 | $S \to (S)$ |
| (01, S, | $ ) | Accept | |

(()) ⇑ (S) ⇑ S

goto 3 (S)

goto 1 S

# DFA dei prefissi viabili / Automa Canonico LR(0)




- da questo DFA (che non è esattamente un DFA!) si ricava la tabella di parsing LR(0)
- ma come si ricava questo DFA a partire dalla grammatica? (lo vedremo tra poco)

---

Le operazioni shift/reduce devono far sì che rimanga in cima alla pila un prefisso di una parte dx di una produzione!
Nel suo complesso, la pila deve contenere un prefisso di derivazione canonica dx!

---

Nell'esempio:

- appena sulla pila compare ")", allora dobbiamo fare una reduce
  Quale reduce fare, dipende dalla "storia", ovvero dallo stato su cui siamo finiti nel DFA

# Prefisso Viabile

**Def(1)** Un prefisso viabile è una stringa $\in (T \cup NT)^*$ che può apparire sulla pila di un parser bottom-up per una computazione che accetta un input.

**Def(2)** (su grammatica G libera)

Una stringa $\gamma \in (T \cup NT)^*$ è un prefisso viabile per G se esiste una derivazione rightmost

$$S \Rightarrow^* \delta A y \Rightarrow \delta \alpha \beta y = \gamma \beta y$$

per qualche $y \in T^*$, $\delta \in (T \cup NT)^*$ e per una produzione $A \to \alpha\beta$. Inoltre S è un prefisso viabile per definizione.

Un prefisso viabile è completo se $\beta = \varepsilon$; in tal caso $\alpha$ è detta maniglia (handle) per $\gamma y$,

(ovvero mi accorgo alla pila trovo $\alpha^R$ e posso fare una reduce)

---

Come fare a:

(i') riconoscere le maniglie sulla cima della pila e ridurle?

(ii') scegliere, tra più riduzioni, solo quelle che producono sulla pila un nuovo prefisso viabile?

(iii') scegliere spostamenti che completino i prefissi viabili sulla pila e facciano comparire una nuova maniglia?

**Teorema** Data G libera, i prefissi viabili di G costituiscono un ling. regolare e può essere descritto con un DFA.

$\Rightarrow$ Il parser (cioè un PDA) può consultare il DFA dei prefissi viabili (ovvero la tabella di parsing) per decidere cosa fare

- se la pila contiene un prefisso viabile completo, allora il parser riduce;
- se la pila contiene un prefisso viabile incompleto, allora il parser shifta;
- se la pila non contiene un prefisso viabile, allora errore.

A seconda di come è fatto il DFA, il parser può risultare deterministico o meno, (eventualmente sfruttando informazioni di look-ahead e sui follow dei nonterminali per ottenere determinismo)

Anche se, concettualmente, ad ogni modifica della pila il DFA la riscandisce da capo per determinare come sia fatto il prefisso viabile corrente, questo non è necessario perché ogni prefisso di un prefisso viabile è un prefisso viabile!

La pila viene modificata in 2 modi:

1) Shift: la pila passa da $\$\gamma$ a $\$\gamma x$. In tal caso il DFA si trova nello stato s dopo aver elaborato $\$\gamma \Rightarrow$ basta far ripartire il DFA da s con input x

2) Reduce $A \to \alpha$: la pila passa da $\$\gamma\alpha$ a $\$\gamma A$. In tal caso il DFA si trova nello stato s dopo aver elaborato $\$\gamma\alpha$; non c'è bisogno di far ripartire il DFA dalla base della pila; basta ripristinare lo stato in cui si trovava subito prima di elaborare il primo simbolo di $\alpha$ e fornirgli il simbolo A in input.

$\Rightarrow$ mi serve lo stack degli stati del DFA!

(anzi, mi basterebbe solo lo stack degli stati, ma noi usiamo anche lo stack dei simboli per ragioni di didattica)

Uno stato del DFA dei prefissi viabili (chiamato automa canonico LR(0)) è costituito da un insieme di item LR(0)

ITEM LR(0): è una produzione con indicata, con un punto, una posizione della sua parte destra

Es: $A \to XYZ$ genera 4 item

$A \to \cdot XYZ$

$A \to X \cdot YZ$

$A \to XY \cdot Z$

$A \to XYZ \cdot$

Il punto "·" indica quanta parte della produzione è già stata analizzata

- se $A \to \alpha \cdot \beta$ è nello stato del DFA in cima alla pila, allora vuol dire che $\alpha$ è sulla pila dei simboli e che ci si aspetta che l'input da leggere contenga (o possa venir ridotto a) $\beta$

- se $A \to \alpha \cdot$ è nello stato del DFA in cima alla pila, allora sulla pila dei simboli c'è la maniglia $\alpha$ e possiamo fare la reduce

# Come costruire l'NFA dei prefissi viabili? (14

Data $G = (NT, T, S, R)$ libera, prendiamo la grammatica aumentata con un nuovo simbolo iniziale $S'$ ed una produzione $S' \to S$

L'NFA dei prefissi viabili di $G$ (primo passo verso la costruzione del DFA!) si ottiene così:

- $[S' \to .S]$ è lo stato iniziale
- dallo stato $[A \to \alpha . X\beta]$ c'è una transizione allo stato $[A \to \alpha X . \beta]$ etichettata $X$, per $X \in T \cup NT$
- dallo stato $[A \to \alpha . X\beta]$, per $X \in NT$ e per ogni produzione $X \to \gamma$, c'è una $\varepsilon$-produzione verso lo stato $[X \to .\gamma]$

Oss: non serve definire degli stati finali, perché l'NFA serve solo come ausilio al parser

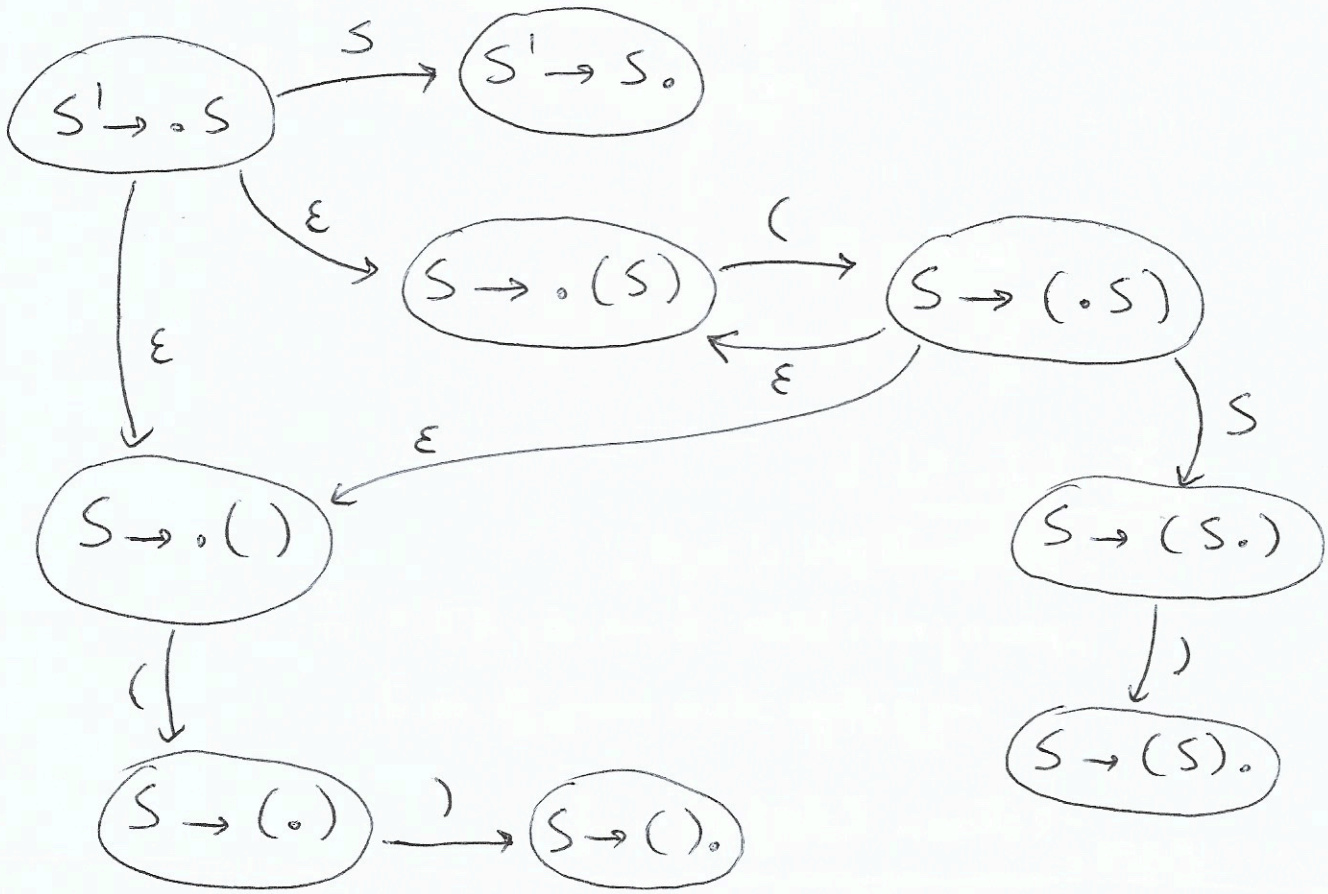
## Esempio

(1) $S' \to S$
(2) $S \to (S)$
(3) $S \to ()$

G

possibili item LR(0) per G

$S' \to .S \mid S.$
$S \to .(S) \mid (.S) \mid (S.) \mid (S).$
$S \to .() \mid (.) \mid ().$




---

## Automa Canonico LR(0)

1) È il DFA che si ottiene dall'NFA dei prefissi viabili mediante la costruzione per sottoinsiemi oppure

2) in modo diretto usando le funzioni ausiliarie
Clos(I) e Goto(I, X)

dove I è un insieme di item e $X \in T \cup NT$

# Costruzione diretta dell'Automa Canonico LR(0)

```
Clos(I) {
    ripeti finché I è modificato {
        per ogni item A → α.Xβ ∈ I
            per ogni produzione X → γ
                aggiungi X → .γ ad I;
                    }
    return I;
    }
```

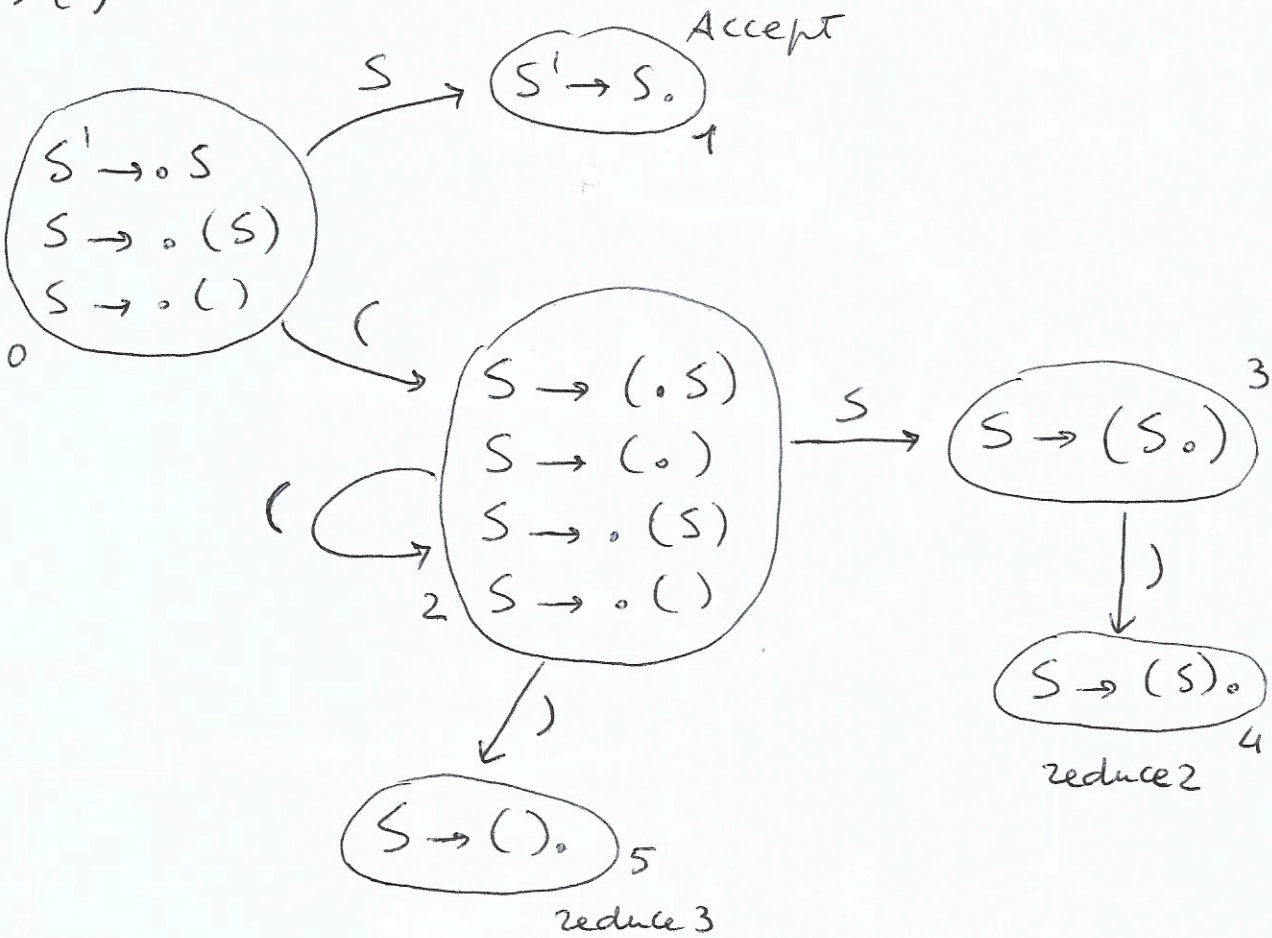
```
Goto(I, X) {
- inizializza J = ∅;
- per ogni item A → α.Xβ ∈ I
      aggiungi A → αX.β a J;
- return Clos(J); }
```

---

Inizializza $\mathcal{S}' = \{ Clos(\{S' \to .S\}) \}$;
Inizializza $\delta = \emptyset$;
Ripeti finché $\mathcal{S}'$ o $\delta$ vengono modificati
    per ogni $I \in \mathcal{S}'$
        per ogni item $A \to \alpha.X\beta \in I$ {
            sia $J = Goto(I, X)$;
            aggiungi $J$ a $\mathcal{S}'$;
            aggiungi $\delta(I, X) = J$ a $\delta$;
            }
return $\mathcal{S}'$ e $\delta$.

# Automa Canonico LR(0)

1 $S' \to S$
2 $S \to (S)$
3 $S \to ()$




$$Clos(\{S' \to .S\}) = \underbrace{\{S' \to .S,\ S \to .(S),\ S \to .()\}}_{I_0}$$

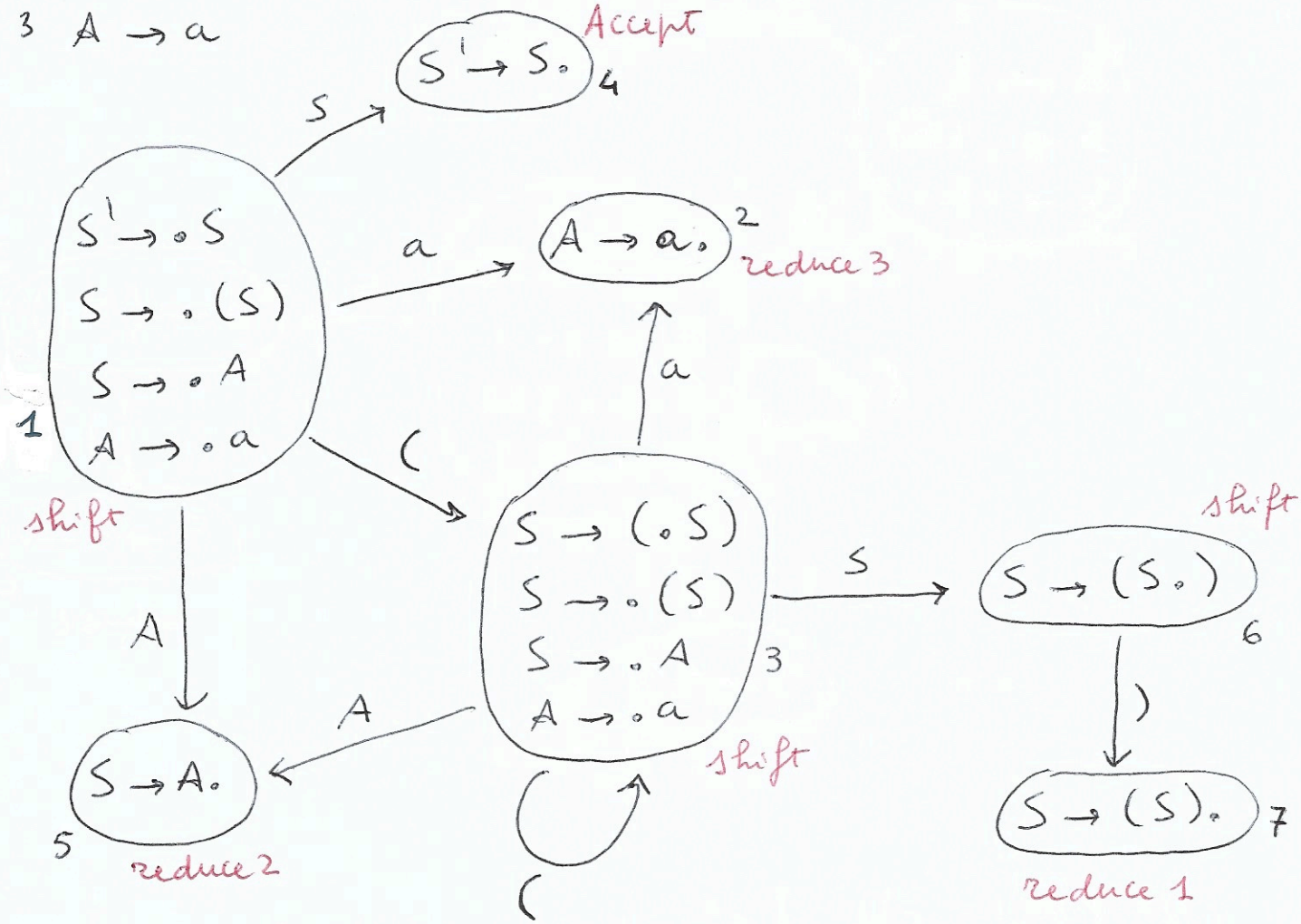
$$Goto(I_0, () = Clos(\{S \to (.S),\ S \to (.)\})$$

$$= \underbrace{\{S \to (.S),\ S \to (.),\ S \to .(S),\ S \to .()\}}_{I_2}$$

# Esempio

0 $S' \to S$
1 $S \to (S)$
2 $S \to A$
3 $A \to a$

$$L(G) = \{ (^n a)^n \mid n \geq 0 \}$$




In questo caso (ed è così sempre per grammatiche LR(0)), possiamo associare una azione ad ogni stato

- stato di shift, come 1, 3 e 6
- stato di reduce, come 2, 5, 7
- stato di accept, come 4

Ma non sarà sempre così ...

# Tabella di Parsing LR

- Matrice bidimensionale M
  - righe = stati dell'automa canonico LR(0) / LR(1)
  - colonne = $\underbrace{T \cup \{\$\}}_{\text{Azioni}} \cup \underbrace{NT}_{\text{GOTO}}$

- M[s, X] contiene le azioni che può compiere un parser LR con s in cima alla pila degli stati e X simbolo in input (o nonterminale)

- Se M[s, X] è "bianca"/vuota, allora ERRORE
- Se M[s, X] contiene più azioni, allora CONFLITTO (il parser non è deterministico)

---

## Caso LR(0)

Per ogni stato s dell'automa canonico LR(0)

- se $X \in T$ e $s \xrightarrow{X} t$ nell'automa LR(0), inserisci shift t in M[s, X]
- se $A \to \alpha . \in s$ e $A \neq S'$, inserisci reduce $A \to \alpha$ in M[s, X] per tutti gli $X \in T \cup \{\$\}$
- se $S' \to S . \in s$, inserisci Accept in M[s, $]
- se $A \in NT$ e $s \xrightarrow{A} t$ nell'automa LR(0), inserisci GOTO t in M[s, A]

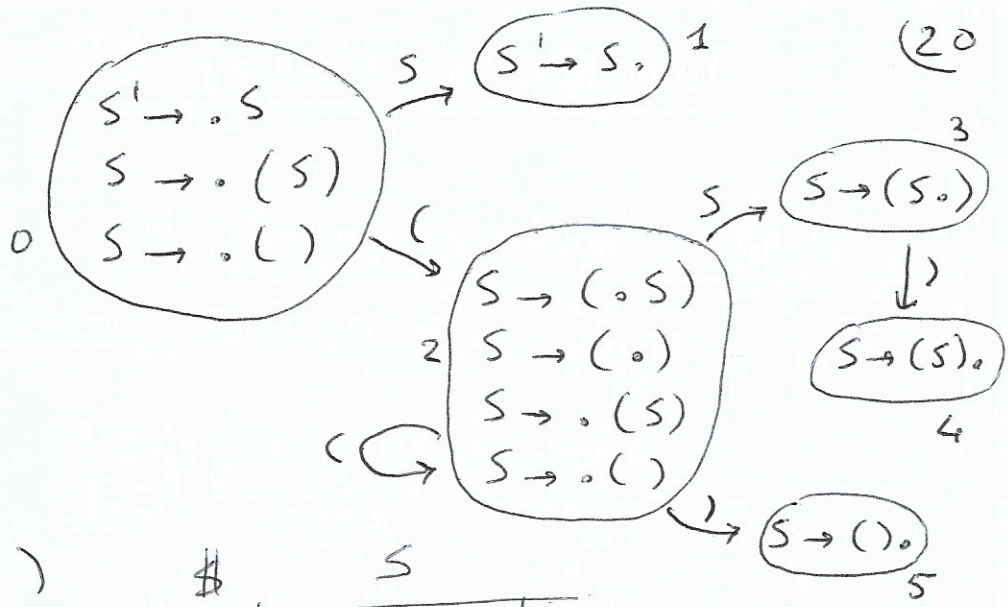
Def Una grammatica è di classe LR(0) se ogni casella nella tabella di parsing LR(0) contiene al più un elemento!

1 $S' \to S$

2 $S \to (S)$

3 $S \to ( )$ ] G

(Esempio di pagina 8)




| | ( | ) | $ | S |
|---|---|---|---|---|
| 0 | s2 | | | g1 |
| 1 | | | acc | |
| 2 | s2 | s5 | | g3 |
| 3 | | s4 | | |
| 4 | r2 | r2 | r2 | |
| 5 | r3 | r3 | r3 | |

Non ci sono conflitti
⇒ G è LR(0)

(0, ε, (())$)

(02, (, ())$)

(022, ((, ))$)

(0225, (()), )$) reduce $S \to ( )$

goto 3 → S

(023, (S , )$)

(0234, (S), $) reduce $S \to (S)$

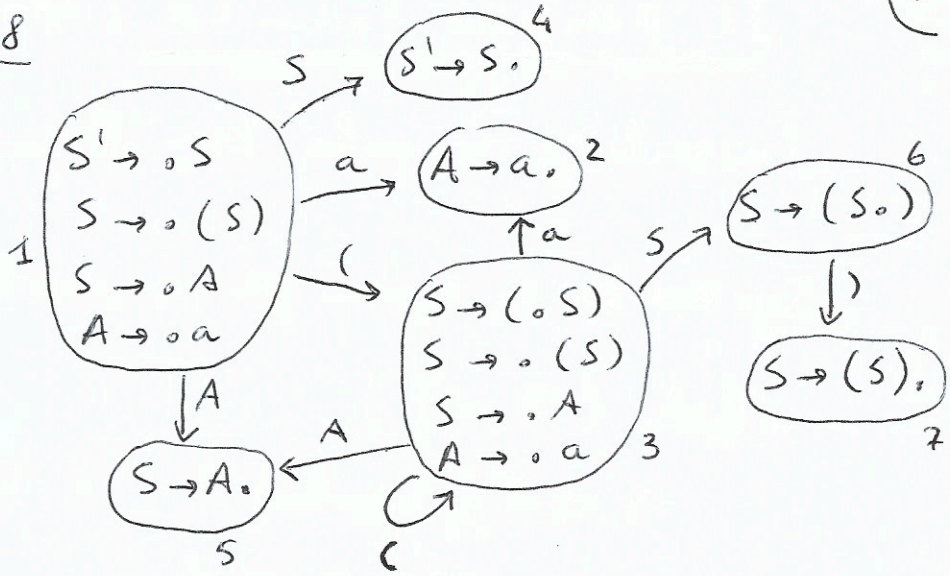
goto 1 → S

(01, S, $) accept

# Esempio di pg 18

0 $S' \to S$
1 $S \to (S)$
2 $S \to A$
3 $A \to a$

] G




| | a | ( | ) | $ | S | A |
|---|---|---|---|---|---|---|
| 1 | s2 | s3 | | | g4 | g5 |
| 2 | r3 | r3 | r3 | r3 | | |
| 3 | s2 | s3 | | | g6 | g5 |
| 4 | | | | acc | | |
| 5 | r2 | r2 | r2 | r2 | | |
| 6 | | | s7 | | | |
| 7 | r1 | r1 | r1 | r1 | | |

Non ci sono conflitti ⇒ G è di classe LR(0)

(1, ε, ((a))$)

(13, (, (a))$)

(133, ((, a))$)

(1332, ((a, ))$) reduce A → a

goto 5

(1335, ((A, ))$) reduce S → A

goto 6

(1336, ((S, ))$)

(13367, ((S), )$) reduce S → (S)

goto 6

(136, (S, )$)

(1367, (S), $) reduce S → (S)

goto 5

(14, S, $)

Accept

# Il generico Parser LR (con solo lo stack degli stati)

- Inizializza la pila con $ $s_0$; % cima della pila a dx
  $s_0$ stato iniziale dell'automa
- Inizializza ic con il primo carattere di input;
- while (true) {
    s = top(pila); % top non rimuove la testa
    case M[s, ic] of
      shift t : { push t sulla pila;
                  avanza ic sull'input; }
      accept : {output ('accept'); break; }
      reduce A→α : { pop |α| stati dalla pila;
        $s_1$ = Top (pila); % $s_1$ contiene B→γ.Aδ
        sia $s_2$ = M[$s_1$, A] % M[$s_1$, A] è goto $s_2$
        push $s_2$ sulla pila;
        output la produzione A→α;
        }
      else errore(); % casella bianca
  }